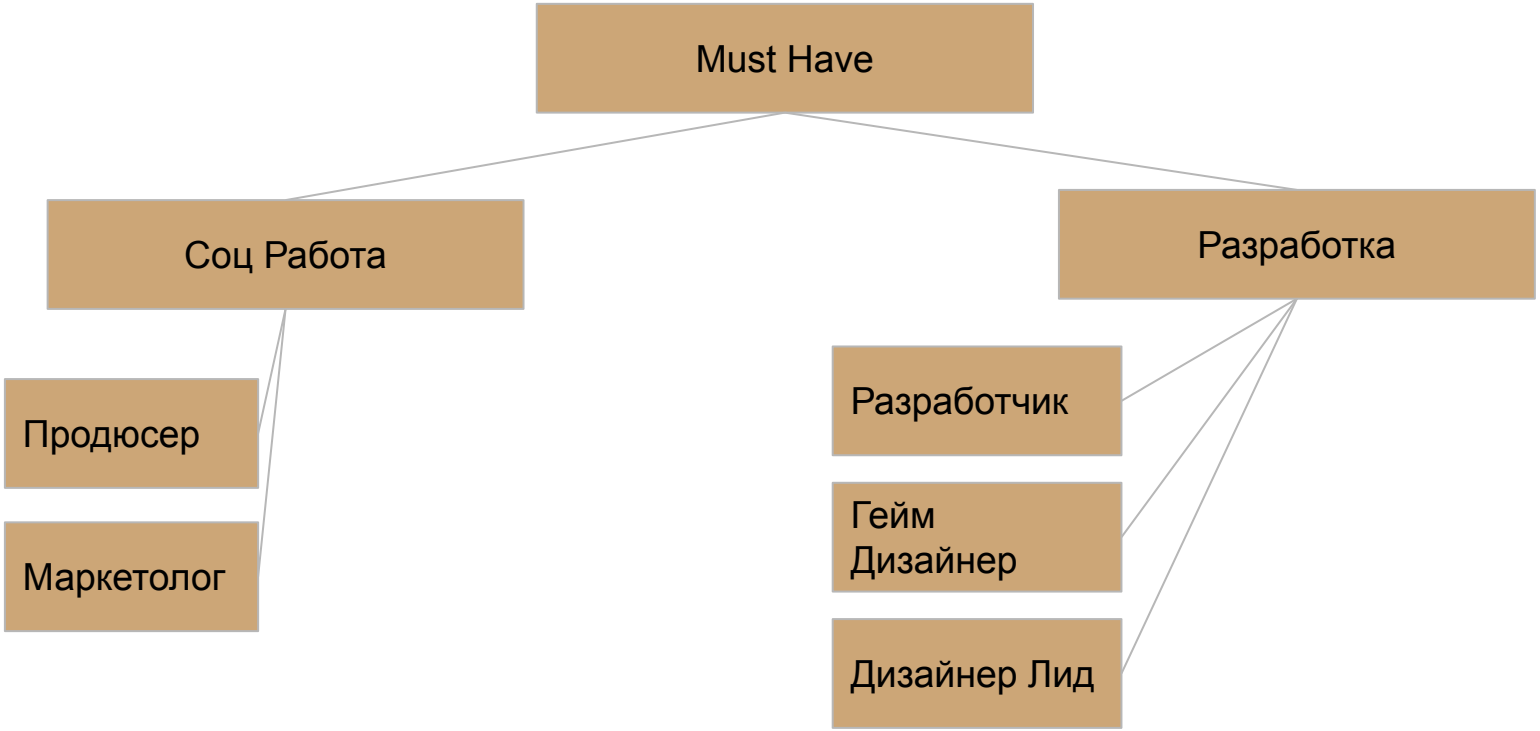
Разработка игр в UE5 на C++

Структура урока

1) Введение
2) Основы C++
3) Создание и настройка проекта
4) Баланс C++ и Блупринов
5) Классы и Макросы

Введение


Must Have
Соц Работа
Разработка
Продюсер
Маркетолог
Разработчик
Гейм Дизайнер
Дизайнер Лид

Разработчик (Developer)

Тема разработчика довольно гигантская, но важно отметить, что 90-95% рынка игр используют два движка: либо Unreal Engine 4 и язык C++, либо Unity и язык C#. Соответственно, если вы планируете быть игровым разработчикам, то вам нужно осваивать один из этих двух движков. На самом деле достаточно классная и редкая для рынка история, если человек умеет работать с обоими движками. Потому что на этом моменте возникают довольно интересные запросы. Иногда в играх бы\Wвают моменты связанные с HTML5. Плюс есть вариант для backend-разработчиков, если у нас игры с серверным взаимодействием, то там достаточно много вариантов решений.

Основные моменты по роли Разработчика:SS

- Пишет код.
- Стоимость в Москве: 120 000 руб.

Гейм Дизайнер(Game Disigner)

Большинство людей, которые хотят пойти в геймдев, зачастую рассматривают позицию геймдизайнера. Это вполне оправданный подход. Геймдизайнер является одной из входных позиций, на которую всегда есть спрос. Самое основное – junior-геймдизайнером можно стать не имея специальных навыков, специального образования.

Чтобы стать геймдизайнером, нужно много играть и иметь достаточно системное мышление и умение писать документацию.

Первое – игровой опыт. У вас должен быть наигрыш исчисляемый в тысячах часов, потому что успешного геймдизайнера, который не играет в игры представить невозможно. Есть два основных подхода, оба из которых достаточно классные:

- У вас есть игра, в которой вами наиграно 10 тысяч часов и больше.
- У вас есть тысяча игр в Steam, в каждую из которых вы играли по несколько часов.

Второе – вы должны системно и чётко излагать свои мысли в письменном виде.

Двух этих критериев достаточно, чтобы стать *Junior*-геймдизайнером. А дальше уже приходить в геймдев и набираться опыта, развиваться.

Важный момент: геймдизайнеры бывают достаточно разными. На крупном проекте могут одновременно работать 10-15 геймдизайнеров, это нормально явление. В *Ubisoft* на проекте типа *Assassin's Creed* может работать 20 геймдизайнеров и каждый будет работать над какой-то своей частью проекта.

Можно выделить четыре категории геймдизайнеров:

1. **Геймдизайнеры-математики.** Те кто занимаются балансом в проекте: подсчётом характеристик юнитов, оружия, сводят таблицы; они же могут заниматься вещами, связанными с монетизацией, расчётом игровой экономики.
2. **Геймдизайнеры-нарративщики.** Люди которые пишут сюжеты, диалоги.
3. **Левел-дизайнеры.** Отдельный подвид геймдизайнеров. Люди, которые придумывают и собирают уровни в играх.
4. **Геймдизайнеры-аналитики.** Люди, которые занимаются разбором продуктов конкурентов, а также занимаются внутриигровыми метриками (например, если у нас free-to-play проект с внутриигровыми платежами, то обязательно есть геймдизайнеры, которые следят за тем, как изменения каких-то параметров игры сказываются на её доходах).

Для *Junior*-геймдизайнеров входная зарплата 50 000 — 60 000 руб.

Геймдизайнеры достаточно быстро растут в зарплатах. Примерный потолок для не *lead*-геймдизайнера это 120 000 руб по Москве.

Основные моменты по роли геймдизайнера:

- Знает всё про игру.
- Лучший кандидат на замену ПМ, если его нет в команде.
- Разбирается в играх, много играет.
- Может писать ТЗ для разработчиков и дизайнеров.
- Стоимость в Москве: 80 000 руб.

Арт Директор(Дизайн Лид)

Ещё один вид специалистов, без которых сделать проект достаточно сложно – это люди, которые занимаются графикой.

Основные моменты по роли Арт-Директора / Дизайн лида:

- Умеет рисовать сам.
- Умеет работать с аутсорсерами и руководить другими дизайнерами.
- Знает как игра должна выглядеть.
- Стоимость в Москве: 100 000 руб.

https://vc.ru/hr/174739-komandy-razrabotchikov-sostav-i-roli-v-gamedev

Основы C++ Разработки (Game Develop)

Функция main

Пожалуй, самая простая и короткая программа на C++ — это программа, которая ничего не делает. Она выглядит так:

```
int main() {

    return 0;

  }
```

Основы C++ Разработки

Здесь определяется функция с именем main, которая не принимает никаких аргументов (внутри круглых скобок ничего нет) и не выполняет никаких содержательных команд. В каждой программе на C++ должна быть ровно одна функция main — с неё начинается выполнение программы.

У функции указан тип возвращаемого значения int (целое число), и она возвращает 0 — в данном случае это сообщение для операционной системы, что программа выполнилась успешно. И наоборот, ненулевой код возврата означает, что при выполнении возникла ошибка (например, программа получила некорректные входные данные).

Для функции main разрешается не писать завершающий return 0, чем мы и будем пользоваться далее для краткости. Поэтому самую короткую программу можно было бы написать вот так:

```
int main() {


}
```

Основы C++ Разработки

Hello, world!

Соблюдая традиции, напишем простейшую программу на C++ — она выведет приветствие в консоль:

```
#include <iostream>

int main() {

    std::cout << "Hello, world!\n";

    return 0;

}
```

Разберём её подробнее.

Основы C++ Разработки

Комментарии

Комментарии — это фрагменты программы, которые игнорируются компилятором (переводит код программы в машинный код) и предназначены для программиста. В C++ есть два вида комментариев — однострочные и многострочные:

```
int main() {  // однострочный комментарий продолжается до конца строки
/* Пример
   многострочного
   комментария */
   }
```

Основы C++ Разработки Переменные

Любая содержательная программа так или иначе обрабатывает данные в памяти. Переменная — это именованный блок данных определённого типа. Чтобы определить переменную, нужно указать её тип и имя. В общем виде это выглядит так:

```
Type name;
```

где вместо Type — конкретный тип данных (например, строка или число), а вместо name — имя переменной. Имена переменных *должны состоять* из латинских букв, цифр и знаков подчёркивания и не должны начинаться с цифры. Также можно в одной строке определить несколько переменных одного типа:

```
Type name1 = value1, name2 = value2, name3 = value3;
```

Основы C++ Разработки Переменные

Например:

```
#include <string>  // библиотека, в которой определён тип std::string

int main() {

    // Определяем переменную value целочисленного типа int

    int value;

    // Определяем переменные name и surname типа std::string (текстовая строка)

    std::string name, surname;

}
```

Основы C++ Разработки Переменные

Важно понимать, что тип остаётся с переменной навсегда. Например, присвоить целочисленной переменной строку не получится — это вызовет ошибку компиляции:

```
int main() {
    int value;
    value = 42;  // OK
    value = "Hello!";  // ошибка компиляции!
}
```

Основы C++ разработки

Ссылки, указатели, константность

Ссылки позволяют вводить псевдонимы для переменных. Указатели — это самостоятельные типы данных: которые могут хранить адреса других переменных в памяти. Ключевое слово `const` позволяет подчеркнуть, что переменная используется только для чтения. Часто оно используется совместно с объявлением ссылок и указателей.

Основы C++ разработки

Ссылки, указатели, константность (Копии переменных)

```cpp
#include <iostream>
#include <string>
  int main() {
      std::string s1 = "Elementary, my dear Watson!";
      std::string s2 = s1;
      s1.clear();  // s2 никак не изменится
      std::cout << s1 << "\n";  // пустая строка
      std::cout << s2 << "\n";  // Elementary, my dear Watson!
  }
```

Основы C++ разработки

Ссылки, указатели, константность (Ссылки)

```
#include <iostream>
int main() {
    int x = 42;
    int& ref = x;  // ссылка на x
    ++x;
    std::cout << ref << "\n";  // 43
}
std::cout << s2 << "\n";  // Elementary, my dear Watson!
}
```

Ссылки, указатели, константность (Указатели)

```
int main() {
    int x = 42;
    int* ptr = &x;  // сохраняем адрес в памяти переменной x в указатель ptr

    ++x;  // увеличим x на единицу
    std::cout << *ptr << "\n";  // 43
}
```

Ссылки, указатели, константность (Указатели)

```
int main() {
    int x = 42;
    int* ptr = &x;  // сохраняем адрес в памяти переменной x в указатель ptr

    ++x;  // увеличим x на единицу
    std::cout << *ptr << "\n";  // 43
}
```

Основы C++ разработки

Ссылки, указатели, константность (Константность)

```
#include <iostream>
int main() {
    const int c1 = 42;  // эта константа известна в compile time
    int x;
    std::cin >> x;
    const int c2 = 2 * x;  // значение становится известным только в runtime
    c2 = 0;  // ошибка компиляции: константе нельзя присвоить новое значение
}
```

Основы C++ Разработки

Ссылки, указатели, константность (Функции)

```
int Sum(int a, int b) {

// в заголовке функции указывается тип возвращаемого значения и типы аргументов

    return a + b;

}
```

Основы C++ Разработки

ООП(объектно ориентированное программирование) Классы

```
1.  class Time {
2.  private:
3.      int hours;
4.      int minutes;
5.      int seconds;
6.  public:
7.      Time(int h, int m, int s);  // объявляем конструктор
8.      // Объявляем три функции для чтения полей:
9.      int GetHours() const;
10.     int GetMinutes() const;
11.     int GetSeconds() const;
12. };
```

Основы C++ Разработки

ООП(объектно ориентированное программирование) Классы

```
1.  class Time {
2.  private:
3.      int hours;
4.      int minutes;
5.      int seconds;
6.  public:
7.      Time(int h, int m, int s);  // объявляем конструктор
8.      // Объявляем три функции для чтения полей:
9.      int GetHours() const;
10.     int GetMinutes() const;
11.     int GetSeconds() const;
12. };
```

Основы C++ Разработки

ООП(объектно ориентированное программирование) Классы

```
Time::Time(int h, int m, int s) {
    if (s < 0 || s > 59) {
        // обрабатываем ошибочные секунды
    }
    if (m < 0 || m > 59) {
        // обрабатываем ошибочные минуты
    }
    if (h < 0 || h > 23) {
        // обрабатываем ошибочные часы
    }
    hours = h;
    minutes = m;
    seconds = s;
}
int Time::GetHours() const {
    return hours;
}
int Time::GetMinutes() const {
    return minutes;
}
int Time::GetSeconds() const {
    return seconds;
}
```

```
Time::Time(int h, int m, int s) {
    if (s < 0 || s > 59) {
        // обрабатываем ошибочные секунды
    }
    if (m < 0 || m > 59) {
        // обрабатываем ошибочные минуты
    }
    if (h < 0 || h > 23) {
        // обрабатываем ошибочные часы
    }
    hours = h;
    minutes = m;
    seconds = s;
}
int Time::GetHours() const {
    return hours;
}
int Time::GetMinutes() const {
    return minutes;
}
int Time::GetSeconds() const {
    return seconds;
}
```

ООП Наследование

```
class A {
private:
    int x;

public:
    void Func1();
    void Func2();
};
```

```
class B: public A {
private:
    int y;

public:
    void Func2();
    void Func3();
};
```

```
int main() {
    B b;
    b.Func1();  // унаследована от A
    b.Func2();  // переопределена в классе B
    b.A::Func2();  // версия Func2 из класса A
    b.Func3();  // определена в классе B
}
```

https://academy.yandex.ru/handbook/cpp

Основы C++

UE предоставляет два метода для создания элементов геймплея — C++ и Blueprint. C++ программисты добавляют основные блоки геймплея, таким образом, чтобы дизайнеры (тут имеется ввиду левел-дизайнер, а не художник) с помощью этих блоков мог создавать свои элементы геймплея для отдельного уровня или всей игры. В таком случае, программисты работают в своем (своей) любимой IDE (например — MS Visual Studio, Xcode), а дизайнер работает в Blueprint редакторе UE.

API геймплея и фреймворк классов полностью доступны из обоих систем. Обе системы можно использовать по отдельности, но используя их вместе вы получаете более мощную и гибкую систему. Это значит, что лучшей практикой будет слаженная работа программистов, которые создают основы геймплея и левел-дизайнеров, которые используют эти блоки для создания увлекательного геймплея.

С учетом всего вышесказанного, далее будет рассмотрен типичный рабочий процесс программиста C++, который создает блоки для дизайнера. В этом случает вы должны создать класс, который в дальнейшем будет расширен с помощью Blueprint, созданного дизайнером или другим программистом. В этом классе мы создадим различные свойства (переменные), которые сможет задать дизайнер. На основе этих заданных значений, мы собираемся извлечь новые значения созданных свойств. Данный процесс очень прост благодаря инструментам и макросам, которые мы предоставляем для вас.

Основы C++

Самое первое что требуется сделать это воспользоваться *мастером классов* (class wizard) предоставляемый UE, для создания базы будущего C++ класса, который в дальнейшем будет расширен с помощью Blueprint.

После того как вы создадите класс, мастер генерирует файлы и откроет IDE, таким образом что вы сразу можете начать редактировать его.

Мастер классов генерирует класс с методами *BeginPlay()* и *Tick()*, со спецификатором *перегрузки (override)*. Событие *BeginPlay()* происходит когда Actor входит в игру, в состоянии разрешённом для игры (*playable state*). Хорошей практикой является инициирование геймплей-кода вашего класса в этом методе. Метод *Tick()* вызывается каждый кадр с параметром, который равен времени, прошедшему с последнего своего вызова. В этом методе должна содержаться постоянно повторяющаяся логика. Если у вас она отсутствует, то лучше всего будет убрать данный метод, что немного увеличит производительность. Если вы удалили код данного метода, убедитесь что вы так же удалили строку в конструкторе класса, которая указывает, что *Tick()* должен вызываться каждый кадр

Создание свойств, отображающихся в редакторе | Классы и Макросы

1)

```
UCLASS()

class AMyActor : public
AActor

{

  GENERATED_BODY()


  UPROPERTY(EditAnywhere)

  int32 TotalDamage;

  }
```

2)

```
UPROPERTY(EditAnywhere, Category="Damage")

int32 TotalDamage;
```

3)

```
UPROPERTY(EditAnyway, BlueprintReadWrite, Category="Damage")

int32 TotalDamage;
```

https://docs.unrealengine.com/5.0/en-US/unreal-engine-uproperties/

https://habr.com/ru/post/254931/